Today

- Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:
Cache Read

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

Direct-Mapped Cache Simulation

M = 16 bytes (4-bit addresses), B = 2 bytes/block, S = 4 sets, E = 1 Blocks/set

Address trace (reads, one byte per read):
0 [0000], miss
1 [0001], hit
7 [0111], miss
0 [0000], miss

Tag Block
Set 0 1 0 M[0-1]
Set 1 1 0 M[2-3]
Set 2 1 0 M[4-5]
Set 3 1 0 M[6-7]

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

If tag doesn’t match: old line is evicted and replaced

If tag doesn’t match: old line is evicted and replaced

Find set
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

Compare both

Valid? + match: yes = hit

Block offset

2-Way Set Associative Cache Simulation

M=16 byte addresses, B=2 bytes/block, S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0 0[0000], miss
1 0[0011], hit
7 0[0111], miss
8 0[1000], miss
0 0[0000], hit

Set 0

v Tag Block
1 00 M(0-1)
1 10 M(8-9)

Set 1

v Tag Block
1 01 M(6-7)
0 00

What about writes?

- Multiple copies of data exist:
  - L1, L2, L3, Main Memory, Disk
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- Typical
  - Write-through + No-write-allocate
  - Write-back + Write-allocate

Intel Core i7 Cache Hierarchy

Processor package

- L1 i-cache and d-cache: 32 KB, 8-way, Access: 4 cycles
- L2 unified cache: 256 KB, 8-way, Access: 10 cycles
- L3 unified cache: 8 MB, 16-way, Access: 40-75 cycles

Block size: 64 bytes for all caches.

Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses)
  - 1 = hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - Can be quite small (e.g., < 1%) for L2, depending on size, etc.
- Hit Time
  - Time to deliver a line in the cache to the processor
  - Includes time to determine whether the line is in the cache
  - Typical numbers:
    - 4 clock cycle for L1
    - 10 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
  - Typically 50-200 cycles for main memory (Trend: increasing!)
Let’s think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider: cache hit time of 1 cycle
    miss penalty of 100 cycles

- Average access time:
  97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
  99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

- This is why “miss rate” is used instead of “hit rate”

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

Rows/Columns Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

32 B = 4 doubles

```c
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

32 B = 4 doubles

Ignore the variables sum, i, j

Rows/Columns Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

32 B = 4 doubles

```c
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

32 B = 4 doubles

Ignore the variables sum, i, j

Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
The Memory Mountain

- **Read throughput** (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

- **Memory mountain**: Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.

---

Memory Mountain Test Function

```c
long data[MAXELEMS]; /* Global array to traverse */
/* test - Iterate over first "elems" elements of *
* array "data" with stride of "stride", using *
* using 4x4 loop unrolling. *
*/
int test(int elems, int stride) {
    long i, sx2 = stride * 2, sx3 = stride * 3, sx4 = stride * 4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;
    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

Call `test()` with many combinations of `elems` and `stride`.
For each `elems` and `stride`:
1. Call `test()` once to warm up the caches.
2. Call `test()` again and measure the read throughput (MB/s)

---

Miss Rate Analysis for Matrix Multiply

- **Assume**:
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
  - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows

- **Analysis Method**:
  - Look at access pattern of inner loop

---

Matrix Multiplication Example

- **Description**:
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - O(N^3) total operations
  - N reads per source element
  - N values summed per destination
  - but may be able to hold in register

```c
for (i=0; i<n; i++)  {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Variable `sum` held in register

---

Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
- Rearranging loops to improve spatial locality
- Using blocking to improve temporal locality
Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - for (i = 0; i < N; i++)
    - sum += a[0][i];
  - accesses successive elements
  - if block size (B) > sizeof(a[i]), bytes, exploit spatial locality
    - miss rate = sizeof(a[i]) / B
- Stepping through rows in one column:
  - for (i = 0; i < n; i++)
    - sum += a[i][0];
  - accesses distant elements
  - no spatial locality!
  - miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Matrix Multiplication (jik)

```c
/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Matrix Multiplication (kij)

```c
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Matrix Multiplication (ikj)

```c
/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (i=0; i<n; i++)
            c[i][j] += r * b[k][j];
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Summary of Matrix Multiplication

<table>
<thead>
<tr>
<th></th>
<th>ijk</th>
<th>jik</th>
<th>kij</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misses/iter</td>
<td>1.25</td>
<td>0.5</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **First iteration:**
  - \( n/8 + n = 9n/8 \) misses

- **Afterwards in cache:**
  - (schematic)
Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size C << n (much smaller than n)

Second iteration:
- Again:
  - n/8 + n = 9n/8 misses

Total misses:
- 9n/8 * n^2 = (9/8) * n^3

Blocked Matrix Multiplication

c = (double *) calloc(sizeof(double), n*n);

```c
void mm(double *a, double *b, double *c, int n) {
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        c[i*n+j] += a[i*n+k] * b[k*n+j];
}
```

Cache Miss Analysis

Assume:
- Cache block = 8 doubles
- Cache size C << n (much smaller than n)
- Three blocks fit into cache: 3B^2 < C

First (block) iteration:
- B^2/8 misses for each block
- 2n/B * B/8 = nB/4 (omitting matrix c)

Afterwards in cache (schematic)

Total misses:
- nB/4 * (n/9)^2 = n^3/4B

Blocking Summary

- No blocking: (9/8) * n^3
- Blocking: 1/(4B) * n^3

Suggest largest possible block size B, but limit 3B^2 < C!

Reason for dramatic difference:
- Matrix multiplication has inherent temporal locality:
  - Input data: 3n^2, computation 2n^3
  - Every array elements used O(n) times!
- But program has to be written properly

Blocking Summary

- No blocking: (9/8) * n^3
- Blocking: 1/(4B) * n^3

Suggest largest possible block size B, but limit 3B^2 < C!

Reason for dramatic difference:
- Matrix multiplication has inherent temporal locality:
  - Input data: 3n^2, computation 2n^3
  - Every array elements used O(n) times!
- But program has to be written properly

Cache Summary

- Cache memories can have significant performance impact
- You can write your programs to exploit this!
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.